Advanced Programming

Reexam: EddyCommand and Pubsubscriber

Kasper Passov pvx884

Contents

1 Eddy Parser

1.1 Grammar

A welldefined grammar is not left-recursive and upholds operator precedence. I have made some leftfactorication to the given grammar to remove the left recursion of command. The only operatorions that calls for precedense is the commands macro and repeat, and the given grammar upholds the precendese.

1.1.1 Changes

The command rule repeat has to be leftfactorized as repeat is left-recursive. I have removed the left recursion by changing the grammar of command giving the new grammar:

```
Script ::= Commands

Commands ::= Command Commands
           | epsilon

Command ::= CommandT * i Repeat'
          | CommandT

Repeat' ::= '*' i Repeat'
          | epsilon

CommandT ::= insert
           | del
           | next
           | prev
           | buffer Ident
           | remove
           | macro Ident '{' Script '}'
           | Ident
```

1.2 Parser choice

I chose to use the ReadP parser as i am most comfortable with it and i find the function munch very usefull, with the SimpleParser does not have.

1.2.1 Ident

To ensure there are at least one character in the ident i made a helper function contains String -¿ String -¿ Bool, that checks if the two strings has one element in common.

```
ident = do skipSpaces
           cs <- munch(`elem` allowed)
           if checkIdents cs && contains cs ([’a’..’z’]++[’A’..’Z’])
              ...
        where allowed = [’a’..’z’]++[’A’..’Z’]++[’0’..’9’]++"_./*?"
              checkIdents s = s ‘notElem‘ reservedKeywords
```

1.2.2 charN

charN allows any of the first 256 characters of the Unicode character set into the insert, and consumes characters until the length described by i is hit.

1.2.3 Integer

I made sure to return the integer as an integer and not a string, so it could easily be read by charN

1.2.4 commandTerm

i moved repeat to its own Nonterminal as describer by the grammar. the rest of the Terminals are unambiguous and without precedence and can as such be put into one Nonterminal.

1.2.5 edrepeat

to ensure a nested repeat is caught and parsed correctly the edrepeat function calls itself, and has the possibility to return without consuming anything.

```
edrepeat ct = do skstring "*"
                 i <- integer
                 edrepeat $ Repeat i ct -- edrepeat is run agein if we have nested repeats
             +++ return ct
```

1.3 Testing

The file ParserTest tests the basic functionality of the Parser, with some edge-cases put in manualy. It is nowhere near complete and i cannot judge the program to be correct on the base of my testing. For this i would need some kind of automatet testing. My testing works by ziping the parser result of some manualy created test with the expected output defined in resultStrings.

1.4 Assesment

Based on my very limited testing, i cannot confidently say a lot about correctness. The Parser is capable of producing useable Abstract Syntax Trees and the general cases seams to work, but there proberly excists some edge cases where the parsing is wrong.

2 Eddy Interpreter

2.1 EddyState Type and monad

I included three things in my State. A list of tuple to represent the buffers and the name of the buffers. The name of the active buffer, and a list of macronames and the matching script. I chose to use a list of the form (key, data), so i could use the buildin library funktions for association lists. Macros are included at the toplevel of the state to make from accessable to all the buffers.

```
data EddyState = EddyState { buffers :: [(BufferName, Buffer)],
                             activeBuffer :: BufferName,
                             macros :: [(MacroName, Script)] }
                 deriving(Show, Eq)

newtype EddyCommand a = EC { runEddy :: EddyState -> (a, EddyState) }
```

I allowed EddyCommand to modify all of the context, as there are no parts of the EddyState that are static. It was possible to create one run that only allows modification on the active buffer that could be used by some of the commands, but i decided ageinst it as i belive it would complicate the code to much.

2.2 Implementation

Very few parts of the implementation is interesting, as they are implementet as

2.2.1 Line Manipulation

The only interesting part of the line manipulation is the fact that the placement of the lines in the first buffer, is reversed. So that the head of the list is the last line. So the lists

```
(["2","1"],["3"])
```

should create the lines

```
1
2
3
```

With this in mind the rest of the line manipulation is straightforward.

2.2.2 Buffer Manipulation

buffer To change the active buffer the only thing needed is change the activeBuffer string in the EddyState. If the chosen buffer is not the list of buffers a new one is created.

remove buffer The function removeBuffer filters out any instances of the buffer by the given name (unless it is the *scratch* buffer) and sets the State of the buffers to the new list without the removed buffer.

2.2.3 Macro Functions

The interesting part of the macro functions are the addition of an already pressent macro. This is done by the addMacro function that creates a list of macros with all macros with the given name filtered out. The new macro is added to this list and set as the new macrolist

```
addMacro :: MacroName -> Script -> EddyCommand()
addMacro mn s = do (EddyState b ab ma) <- getState
                   let newma = filter (\(x,_) -> x /= mn) ma ++ [(mn,s)]
                   setState (EddyState b ab newma)
```

2.3 The type Error

I did not complete this part of the excercise, so as of now it simply ignores any command that would create and error. It should however be easy to implement it from the current Interpreter as i have forwarded every instance that should call an error into the function raiseError. The reason for it not being implemented, is me not knowing how, and prioritizing implementing the things i do know before learning to implement things i do not.

```
-- raiseError :: Error
raiseError :: EddyCommand()
raiseError = do es <- getState --error goes here
                setState es
```

2.4 Testing

The only testing done on the Interpreter is by hand in the enviorment. So i can saftly say there are a lot of cases that it will fail. But i was able to confirm that there are cases where it returns as expectet. The lack of real testing is mainly me trying to build an automatet tester via quickcheck, but having to throw it away because of time limitations and its current state being without any functionality.

2.5 Assesment

The gross lacking of tests make this implementation very shaky. I do belive this is implementation can run most wellformed AST's given by the parser as I found the basic functionality working. I am basing this on the testing with different Scripts I did in the environment, but with no testing framework or even a small script with manually writen tests i cannot call it correct.

3 Pubsub

3.1 Implementation

I chose not to use OTP as i fell more comfortable building the server myself, where there are as little hiden code as possible. I realise this creates uglier and more code, but I generaly fell the quality of my implementation is improved by this choice.

3.1.1 Node

Ones startet, each node in the network has 3 empty lists of tuples. The first saves the nodes subs in tuples of process ids and filters [S, F]. This is done so the node can find and access each subscriber and the filter for that subscriber. The second list of tuples is a list of messages this node has recieved, and the uniqe id of that message [E, Ref]. The last list of tuples contains and the process that cast the error [E, S].

3.1.2 The Error list

The error list is populated when a filter fails on some input. This is firstof to uphold the robustness of the network so a filter throwing an error does not halt the network. Secondly it is saved so the users or adminstrators have some way of discovering why the nodes does not send the messages to the subscribers. For this I extended the API to contain an errors/1 function that takes a process id and returns that nodes list of filter errors.

3.1.3 Classes of Mistakes

The helper function tryFilter/4 is called when a node recieves a message and all subscribers Filter functions are to called on it. The helperfunction tries to run the filter on the element and returns the filters response if its returntype is a boolean. If the filter function throws some kind of error it is caught by the filters try/catch, the second type of bad filter i could think of, is one where the filter succeds, but does not return a boolean. In this case i have chosen to return false and inform the node of the filter not returning as expected. So the classes of mistakes the implementation can handle, is the filter throwing exceptions and type errors. One type of filter error i do not catch is that of a filter function that creates an infinite loop. In such a case the node will be stuch as it does not know it is in an infinite loop and it waits for it to return. Unfortuantly I though of this problem to late for me to implement the solution The solution i would have used for this, would be some kind of timeout running parallel with the filter function, that after a certain amount of time sends a halt messages, that forces the node to treat the filterfunction as if it returned false.

3.1.4 Subscribtion loops

One potential weakness of the network is nodes subscribers creating loops between them. This could potentialy create infinite loops where a message is bounced between a number of subscribers when filters does not stop the message. The implementation of my network stops this by binding an uniqe reference to every message. Everytime a node recieves a message, it checks its message list of any references equal to that recieved. If one of those is found the message is not forwarded or saved. Assuming erlang:make_ref() (practicaly) creates an uniqe id, an already send message should never be saved or resend. This solution means it is possible to publish identical messages throughout the sytem without the messages being stopped, as all new messages have a new identifier. make_ref recycles ids after 2$\hat{8}$2 calls, meaning there are some posibility that a messages will be lost if enough messages is called. This number is large enough for it to be unique enough for practical purposes, if not in theory.

3.2 Testing

I use the lightweight testing framework EUnit to test my erlang implementation, and the analysing tool dialyzer. Dialyzer throws passes it with no warnings. I have made 8 manual tests for EUnit that can be run with pubsub:test(), this tests the general functionality of the API, and everything seams to work as expected. The demonstration asked for in the assignment can be run using pubsub:demonstration() and returns

```
2>pubsub:demonstration().
{[emc2,"Hello",{tick,tock}],["Hello",4]}
```

3.3 Assesment

I belive this to be the best of my solutions. I cannot say the implementation is correct. as i have made no automatic testing for testing larger system with a large amount of messages. This could mean there are critical flaws in the system i have not seen or accountet for. The one i do know is pressent is the looping filter function that can shut down the network. But this problem i belive can be patched in version 0.2. Overall i belive a fix to the known problem and more thorough testing is needed for me to satistied with my solution of this pubsub server. I do it could be deployed with some monitoring, as my tests shows the API seams to implemented and return values correctly.

4 General Assesment

Based on my lacking tests of the Parser and the practicly untested interpreter, the theme of this rapport is me saying very little on the correctness of the Eddy implementation. My assesments of the individual parts of the assignment is as follows:

- Parser Medium: It seams to work and parse as describer by the assignment, but it lacks testing.
- Interpreter Low: *Very* limited testing, a non excisting error handling, and a weak grasp of user defined monads.
- Erland Medium-High: The best testing of the 3, but still insificant.

Overall i belive this to be an average implementation, and a weak testing, but it does show some insight into the workings of Haskell and Erlang.

5 Appendix

EddyAst.hs

```haskell
module EddyAst where

type Script = [Command]
data Command = Ins String
             | Del
             | Next
             | Prev
             | Repeat Int Command
             | Buffer BufferName
             | Remove
             | Macro MacroName Script
             | Call MacroName
             deriving (Eq, Show)
type BufferName = String
type MacroName = String
```

EddyParser.hs

```
--
-- Skeleton for Eddy parser
-- To be used at the re-exam for Advanced Programming, B1-2013
--

----------------------------
-- Student: Kasper Passov --
-- KU ID:   pvx884        --
----------------------------

{-# OPTIONS_GHC -Wall -fno-warn-unused-do-bind #-}

module EddyParser
  ( Error
  , EddyParser.parse
  , parseFile
  , contains
  ) where

import Text.ParserCombinators.ReadP
import Data.Char
import EddyAst

data Error = Error deriving (Show, Eq)

-----------------------------
--           API           --
-----------------------------

parseFile :: FilePath -> IO (Either Error Script)
parseFile f = do s <- readFile f
                 return $ EddyParser.parse s

parse :: String -> Either Error Script
parse input = case output of
    (p, ""):_ -> Right p
    _         -> Left Error
  where output = readP_to_S (do
                  sc <- script
                  skipSpaces
                  eof
                  return sc) input

-----------------------------
```

```haskell
-- Internal Implementation --
-----------------------------

script :: ReadP Script
script = commands

commands :: ReadP [Command]
commands = many command

command :: ReadP Command
command = do ct <- commandTerm
             rest ct
          where
             rest ct = edrepeat ct
                   +++ return ct

edrepeat :: Command -> ReadP Command
edrepeat ct = do skstring "*"
                 i <- integer
                 edrepeat $ Repeat i ct -- edrepeat is run agein if we have nested repeats
             +++ return ct

commandTerm :: ReadP Command
commandTerm = insert +++ del    +++ next  +++ prev +++
              buffer +++ remove +++ macro +++ call
            where
              insert = do skstring "i"
                          i <- integer
                          skstring "|"
                          cn <- charN i
                          return (Ins cn)
              del    = do skstring "del"
                          return Del
              next   = do skstring "next"
                          return Next
              prev   = do skstring "prev"
                          return Prev
              buffer = do skstring "buffer"
                          s <- ident
                          return (Buffer s)
              remove = do skstring "remove"
                          return Remove
              macro  = do skstring "macro"
                          st <- ident
                          skstring "{"
                          sc <- script
                          skstring "}"
                          return (Macro st sc)
              call   = do st <- ident
                          return (Call st)

integer :: ReadP Int
integer = do skipSpaces
             n <- many1 (satisfy isDigit)
             return $ read n

charN :: Int -> ReadP String
charN i = do c <- many (satisfy isLatin1)
             if length c == i
                then return c
                else pfail

ident :: ReadP String
ident = do skipSpaces
           cs <- munch(‘elem‘ allowed)
           if checkIdents cs && contains cs ([’a’..’z’]++[’A’..’Z’])
              then return cs
              else pfail
        where allowed = [’a’..’z’]++[’A’..’Z’]++[’0’..’9’]++"_./*?"
              checkIdents s = s ‘notElem‘ reservedKeywords

contains :: String -> String -> Bool
contains []    _ = False
contains (h:t) l = contains’ h l || contains t l

contains’ :: Char -> String -> Bool
contains’ _ []     = False
contains’ e (h:t)  = h == e || contains’ e t

-----------------------------
--          Misc           --
-----------------------------

skstring :: String -> ReadP String
skstring st = skipSpaces >> string st

reservedKeywords :: [String]
reservedKeywords = ["i", "del", "next", "prev", "buffer", "remove", "macro"]
```

ParserTest.hs

```haskell
module ParserTest (test) where

import EddyParser
import EddyAst

testStrings :: [String]
testStrings = ["i 6|World! prev i 5 |Hello" --The two given examples
              ,"buffer world i 2|to i 2|my i 6|World! prev i 25|wonderful, yet terrifyingmac
              ,"    i        9  |   input     " -- arbitrary  whitespaces
              ,"del * 5 * 5" -- nested repeat
              ,"*_A123_*/?." -- symbols for buffer
              ]

resultStrings :: [Either Error Script]
resultStrings = [Right [Ins "World!", Prev, Ins "Hello"] -- right output of examples
                ,Right [Buffer "world", Ins "to", Ins "my", Ins "World!", Prev
                       ,Ins "wonderful, yet terrifying", Macro "top" [Repeat 2 (Repeat 5000
                       ,Call "top" ,Ins "Welcome"]
                ,Right [Ins "   input "]
                ,Right [Repeat 5 (Repeat 5 Del)]
                ,Right [Call "*_A123_*/?."]
                ]
test :: Bool
test = foldl (\ x y -> x && y ) True list
      where list = runTests testStrings resultStrings

runTests :: [String] -> [Either Error Script] -> [Bool]
runTests = zipWith (\x y -> parse x == y)
```

EddyInterp.hs

```
--
-- Skeleton for Eddy interpreter
-- To be used at the re-exam for Advanced Programming, B1-2013
--
----------------------------
-- Student: Kasper Passov --
-- KU ID:   pvx884        --
----------------------------

{-# OPTIONS_GHC -Wall #-}

module EddyInterp (runScript) where

import EddyAst

type Line = String
type Buffer = ([Line], [Line])

data Error = Error deriving(Show, Eq)

data EddyState = EddyState { buffers :: [(BufferName, Buffer)],
                             activeBuffer :: BufferName,
                             macros :: [(MacroName, Script)] }
                 deriving(Show, Eq)

newtype EddyCommand a = EC { runEddy :: EddyState -> (a, EddyState) }

instance Monad EddyCommand where
  return a = EC $ \e -> (a, e)
  a >>= f  = EC $ \e ->
              let (j, en) = runEddy a e in
                runEddy (f j) en

emptyState :: EddyState
emptyState = EddyState { buffers = [("*scratch*", ([],[]))],
                         activeBuffer = "*scratch*",
                         macros = [] }

----------------------------------------
-- Functions to manipulate the states --
----------------------------------------

-- returns the state
getState :: EddyCommand EddyState
getState = EC $ \s -> (s, s)

-- sets the current state to ns
setState :: EddyState -> EddyCommand ()
setState ns = EC $ const ((), ns)

-------------------------
-- Buffer manipulation --
-------------------------

-- returns the buffer with the name bn
getBuffer :: BufferName -> EddyCommand Buffer
getBuffer bn = do (EddyState nbufs _ _) <- getState
                  case lookup bn nbufs of
                       Nothing -> getBuffer "*scratch*"
                       Just b  -> return b

-- adds a buffer b to the bufferlist
addBuffer :: (BufferName, Buffer) -> EddyCommand()
addBuffer (bn,b) = do (EddyState bufs _ ma) <- getState
                      setState(EddyState (bufs++[(bn,b)]) bn ma)

-- removes a buffer unless it is *scratch*
removeBuffer :: EddyCommand()
removeBuffer = do (EddyState b ab ma) <- getState
                  let newb = filter(\(x,_) -> x /= ab) b
                  if ab == "*scratch*"
                    then raiseError
                    else setState(EddyState newb "*scratch*" ma)

-------------------------
-- Active manipulation --
-------------------------

-- returns the active buffer
getActive :: EddyCommand Buffer
getActive = do (EddyState _ ab _) <- getState
               getBuffer ab

-- sets the active buffer to the buffer with the name bn
changeActive :: BufferName -> EddyCommand()
changeActive bn = do (EddyState b _ ma) <- getState
                     case lookup bn b of
                       Nothing    -> addBuffer (bn, ([],[])::Buffer)
                       Just _     -> setState (EddyState b bn ma)
```

```
-- update the active buffer to b
updateActive :: Buffer -> EddyCommand()
updateActive b = do (EddyState bufs bn ma)  <- getState
                    let newb = filter (\(x,_) -> x /= bn) bufs ++ [(bn, b)]
                    setState (EddyState newb bn ma)

------------------------
-- Macro manipulation --
------------------------

-- adds a macro (mn, s) to the current macros
addMacro :: MacroName -> Script -> EddyCommand()
addMacro mn s = do (EddyState b ab ma) <- getState
                   let newma = filter (\(x,_) -> x /= mn) ma ++ [(mn,s)]
                   setState (EddyState b ab newma)

-- runs the macro mn
runMacro :: MacroName -> EddyCommand()
runMacro mn = do (EddyState _ _ ma) <- getState
                 case lookup mn ma of
                   Nothing -> raiseError
                   Just sc -> script sc

------------------------
--   Error function   --
------------------------

-- raiseError :: Error
raiseError :: EddyCommand()
raiseError = do es <- getState --error goes here
                setState es

edrepeat :: Int -> Command -> EddyCommand()
edrepeat 0 _ = do es <- getState -- det her skammer jeg mig over
                  setState es
edrepeat i c = do command c
                  edrepeat (i-1) c

------------------------
--Script Functionality--
------------------------

script :: Script -> EddyCommand()
script []     = do es <- getState
                   setState es
script (c:sc) = do command c
```

```
                   script sc

command :: Command -> EddyCommand ()
command c = case c of
              (Buffer bn)    -> changeActive bn
              (Remove)       -> removeBuffer
              (Macro mn s)   -> addMacro mn s
              (Call m)       -> runMacro m
              (Repeat i co)  -> edrepeat i co
              _              -> command' c

command' :: Command -> EddyCommand ()
command' c = do ab <- getActive
                newb <- case c of
                        (Ins l) -> insertLine l ab
                        (Del)   -> delLine ab
                        (Next)  -> nextLine ab
                        (Prev)  -> prevLine ab
                        _       -> return ab -- Some error
                updateActive newb

-----------------------
-- Line manipulation --
-----------------------

insertLine :: Line -> Buffer -> EddyCommand Buffer
insertLine l (lb, la) = return (l:lb, la)

delLine :: Buffer -> EddyCommand Buffer
delLine (lb, []) = return (lb, [])
delLine (lb, _:la) = return (lb, la)

nextLine :: Buffer -> EddyCommand Buffer
nextLine (lb, []) = return (lb, [])
nextLine (lb, lah:la) = return (lah:lb, la)

prevLine :: Buffer -> EddyCommand Buffer
prevLine ([], la) = return ([], la)
prevLine (lbh:lb, la) = return (lb, lbh:la)


-----------------------
--        API        --
-----------------------

runeddy :: Script -> EddyCommand Buffer
runeddy sc = do script sc
                getActive

runScript :: Script -> Either Error [(BufferName, String)]
runScript sc = case buflist of
                  a -> Right a
                  {- _ -> Left Error -} -- errors are not implementet
      where
    buflist = map (\(bufname, (lprev, lnext)) -> (bufname, unlines ( reverse lprev ++ lnext)
    (_, EddyState buffs _ _ ) = runEddy ( runeddy sc ) emptyState
```

pubsub.erl

```
%%%-------------------------------------------------------------------
%%% @author Ken Friis Larsen <kflarsen@diku.dk>
%%% @copyright (C) 2014, Ken Friis Larsen
%%% @doc
%%% Skeleton for the re-exam for Advanced Programming, B1-2013
%%% Implementation of pubsub server
%%% @end
%%%-------------------------------------------------------------------
%%% Student name: Kasper Passov
%%% Student KU-id: pvx884
%%%-------------------------------------------------------------------
-include_lib("eunit/include/eunit.hrl").
-module(pubsub).
-export([start/0, add_subscriber/3, stop/1, subscribers/1, publish/2, demonstration/0]).



%%%-------------------------------------------------------------------
%%% API
%%%-------------------------------------------------------------------

start() ->
    {ok, spawn(fun() -> psnode([],[],[]) end)}.

add_subscriber(P, S, F) ->
    rpc(P, {S, F, add_sub}).

subscribers(P) ->
    rpc(P, get_subs).

publish(P, E) ->
    Ref = erlang:make_ref(),
    info(P, {{E, Ref}, publish}).

messages(P) ->
    rpc(P, messages).


%%%-------------------------------------------------------------------
%%% API extensions
%%%-------------------------------------------------------------------
% These API extensions have been made to make testing easier

% The stop function kills the given process P. It does not kill or
% message its subscribers
stop(P) ->
    rpc(P, stop).

% Returns all errors the given process has saved. An error happens
% when a filter fails or returns something that is not a bool
errors(P) ->
    rpc(P, errors).
%%%----------------------------------------------------------------------
%%% Internal Implementation
%%%----------------------------------------------------------------------

psnode(Subs, Messages, Errors) ->
    receive
        {From, stop} ->
            reply(From, {stopped, self()});
        {From, errors} -> % return all errors
            reply(From, Errors);
        {Error, S, error} -> % save an error
            psnode(Subs, Messages, Errors ++ [{Error, S}]);
        {From, {S, F, add_sub}} -> % add a subscriber
            case lookupSubs(S, Subs) of
                  none -> reply_ok(From),
                          psnode([{S,F}] ++ Subs, Messages, Errors);
                  _ -> reply_error(From),
                       psnode(Subs, Messages, Errors)
            end;
        {From, get_subs} -> % return subscribers
            reply(From, getAll(Subs)),
            psnode(Subs, Messages, Errors);
        {{E, Ref}, publish} -> % forward message
            case lists:keyfind(Ref, 2, Messages) of
                false -> PassedList = lists:filter(fun({S,F}) -> tryFilter(self(),S,F,E) end
                         lists:foreach(fun({S,_}) -> info(S, {{E, Ref}, publish}) end, Passe
                         psnode(Subs, [{E, Ref}] ++ Messages, Errors);
                _     -> psnode(Subs, Messages, Errors)
            end;
        {From, messages} -> % return all messages
            reply(From, getAll(Messages)),
            psnode(Subs, Messages, Errors);
        _ ->
            erlang:display(end_of_pubserver) % helps me find lost messages
    end.
%%%----------------------------------------------------------------------
%%% Helper Functions
%%%----------------------------------------------------------------------
```

```erlang
tryFilter(From, S, F, E) ->
    try F(E) of
        true -> true;
        false -> false; % function fails
        _ -> info(From, {filter_not_bool, S, error}),
             false %function does not return bool
    catch % Funktion fails on type
    _ : _ -> info(From, {filter_fails, S, error}),
                 false %"Bad filter function"
end.

lookupSubs(_, []) -> none;
lookupSubs(S, [{S, F} | _]) -> F;
lookupSubs(S, [_ | SUBS]) -> lookupSubs(S, SUBS).

getAll([]) -> [];
getAll([{A, _} | Rest]) -> getAll(Rest) ++ [A].

%%%-------------------------------------------------------------------
%%% Demonstration
%%%-------------------------------------------------------------------

% demonstration as describer on page 9 of the exam set
demonstration() ->
    {ok, Niels} = start(),
    {ok, Albert} = start(),
    {ok, Christiaan} = start(),
    {ok, Isaac} = start(),
    {ok, Joseph_Louis} = start(),
    {ok, Johannes} = start(),
    {ok, Euclid} = start(),
    add_subscriber(Albert, Niels, fun(_) -> true end),
    add_subscriber(Isaac, Albert, fun(X) -> hd(tl(X)) == 101 end), % 101 is the character ’
    add_subscriber(Christiaan, Albert, fun(_) -> true end),
    add_subscriber(Euclid, Isaac, fun(X) -> X rem 2 == 0 end),
    add_subscriber(Johannes, Isaac, fun(_) -> true end),
    add_subscriber(Isaac, Joseph_Louis, fun(_) -> true end),
    publish(Euclid, 5),
    publish(Euclid, 4),
    publish(Euclid, point),
    publish(Isaac, "Hello"),
    publish(Christiaan, {tick, tock}),
    publish(Albert, emc2),
    timer:sleep(10),
    {messages(Niels),messages(Joseph_Louis)}.
```

```
%%%-------------------------------------------------------------------
%%% Testing
%%%-------------------------------------------------------------------

% tests start and stop
start_test() ->
    {ok, P} = start(),
    {stopped, P} = stop(P).

% tests publish and messages
publish_one_test() ->
    {ok, P} = start(),
    publish(P, "Winter"),
    publish(P, "is"),
    publish(P, "comming"),
    ["Winter","is","comming"] = messages(P).

% tests adding subscribers
add_sub_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(_) -> true end),
    add_subscriber(A, C, fun(_) -> true end),
    [B, C] = subscribers(A).

% tests publishing to subscribers
publish_subs_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(_) -> true end),
    add_subscriber(A, C, fun(_) -> true end),
    error = add_subscriber(A, C, fun(_) -> true end),
    publish(A, "A is greatest"),
    timer:sleep(10),
    ["A is greatest"] = messages(A),
    ["A is greatest"] = messages(B),
    ["A is greatest"] = messages(C).

%tests filters
publish_filter_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(X) -> X < 4 end),
    add_subscriber(A, C, fun(X) -> X > 2 end),
    publish(A, 2),
    publish(A, 3),
    publish(A, 4),
    timer:sleep(10),
    AM = messages(A),
    BM = messages(B),
    CM = messages(C),
    {[2,3,4],[2,3],[3,4]} = {AM,BM,CM}.

% tests error handling and rebustness
error_function_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, C, fun(_) -> throw(poo) end),
    add_subscriber(A, B, fun(X) -> X end),
    publish(A, "zoo abes throw "),
    timer:sleep(10),
    AE = errors(A),
    [{filter_not_bool,B}, {filter_fails,C}] = AE.

% message is recieved through nested sub
nested_subs_test() ->
    {ok, A} = start(),
    {ok, B} = start(),
    {ok, C} = start(),
    add_subscriber(A, B, fun(_) -> true end),
    add_subscriber(B, C, fun(_) -> true end),
    publish(A, "we can go deeper"),
    timer:sleep(10),
    ["we can go deeper"] = messages(C).

% messages are not continuously send
sub_loop_test() ->
    {ok, A} = start(),
    ok = add_subscriber(A, A, fun(_) -> true end),
    publish(A, "recurring"),
    timer:sleep(20),
    ["recurring"] = messages(A).

%%%-------------------------------------------------------------------
%%% Communication primitives
%%%-------------------------------------------------------------------
```

```
%% synchronous communication

rpc(Pid, Request) ->
    info(Pid, {self(), Request}),
    receive
        {Pid, Response} ->
            Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.

reply_ok(From) ->
    reply(From, ok).

reply_error(From) ->
    reply(From, error).

%% asynchronous communication

info(Pid, Msg) ->
    Pid ! Msg.
```